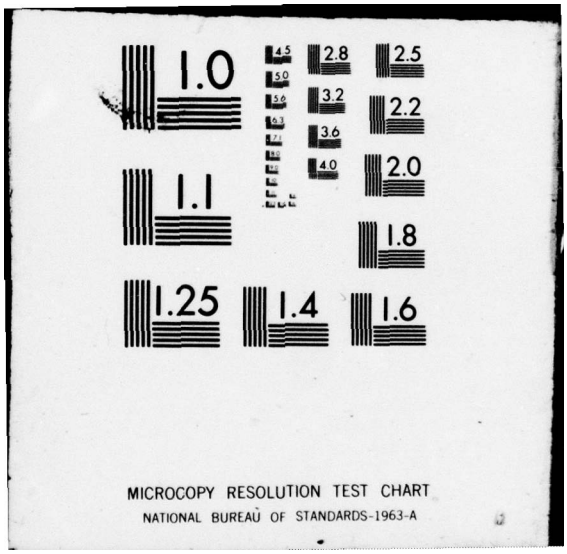
AD-A071 599 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 5/8
ARCHITECTURE FOR HIGHER LEVEL DIGITAL IMAGE PROCESSING.(U)
JUL 78 T J WILLETT DAAG53-76-C-0138
UNCLASSIFIED NL

1 OF 1
AD A071599

END
DATE FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ARCHITECTURE FOR HIGHER

LEVEL DIGITAL IMAGE PROCESSING

July 30, 1978

This is the first quarterly status report on a program for Image Understanding Using Overlays, conducted by Westinghouse for Maryland under Contract DAAG 53-76 C-0138 with the U.S. Army Mobility Equipment Research and Development Command, Fort Belvoir, Va. 22060.

Prepared for

Computer Science Center
University of Maryland
College Park, Maryland 20742

By

Westinghouse Defense and Electronic Systems Center
Systems Development Division
Baltimore, Maryland 21203

LEVEL II

1

6 ARCHITECTURE FOR HIGHER LEVEL DIGITAL IMAGE PROCESSING

July 30, 1978

11 30 Jul 78

12 66 p.

This is the first quarterly status report on a program for Image Understanding Using Overlays, conducted by Westinghouse for Maryland under Contract 15 DAAG 53-76-C-0138 with the U.S. Army Mobility Equipment Research and Development Command, Fort Belvoir, Va. 22060.

9 Quarterly status rept. no. 1.

Prepared for

Computer Science Center
University of Maryland
College Park, Maryland 20742

10 Thomas J. Willett

By

Westinghouse Defense and Electronic Systems Center
Systems Development Division
Baltimore, Maryland 21203

403 028

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

This is the first quarterly status report on a program to investigate various approaches to the design of architecture for higher level digital image processing algorithms, being conducted by the Westinghouse Systems Development Division for the Computer Science Center, University of Maryland. This two-year program is a continuation of a program entitled "Algorithms and Hardware Technology for Image Recognition", which was initiated in 1976. The report was prepared by Mr. Thomas J. Willett. The Westinghouse program manager is Dr. Glenn E. Tisdale.

During the quarter, monthly technical meetings were held at Maryland, which included representatives from the Army Night Vision Laboratory, the University of Maryland, and Westinghouse. Team members from NVL were Dr. George Jones and Mr. John Dehne, and from the University of Maryland, Profs. David Milgram and Azriel Rosenfeld.

The report begins with a review of desired system design goals. This is followed by a description of available microprocessor hardware, a review of LISP approach to the manipulation of list structures, and a preliminary discussion of the processing required to implement relaxation methods of object classification. The report concludes with a description of specific bit-slice processors.

1.0 SYSTEM DESIGN GOALS

Westinghouse, following the Smart Sensor work, has been given the assignment of implementing the Maryland higher level image processing algorithms in a digital processor. The purpose of this section is to describe the system design goals which the hardware and software must meet, goals such as speed, reliability, size, weight, probability of detection, recognition, or false alarm, power consumption, and software requirements. Each is now considered in a separate subsection.

1.1 Speed

The typical image processor has a block diagram like that of Figure 1-1. The preprocessor filters the image (e.g., low pass or median filter) to smooth it and extracts primitives such as edges, amplitudes above a certain threshold, and histograms; it also thins the extracted data to reduce

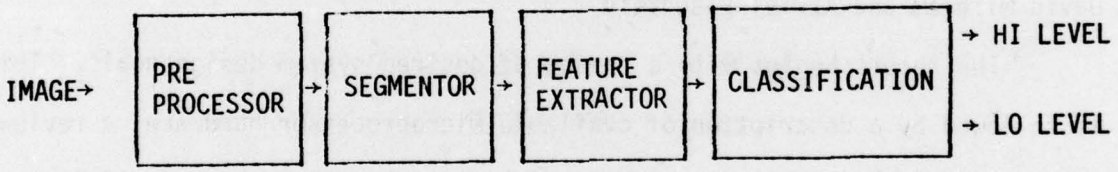


Figure 1-1. Block Diagram of Image Processor

the amount and keep only the most important. The segmentor separates data associated with a specific object, including threshold densities, edges, and functions of them such as texture. The feature extractor then forms pre-classification features such as shape, area, length, curvature, density, etc. The classification process then attempts to identify the collection of objects in the image. Assuming a video frame of 550 x 600 pixels, for a total of 330,000 pixels per frame, the processor would have to process 1 million pixels per second to achieve a frame rate of 3 per second. Real time frame rates are 30 per second, or 10 megapixels per second. If we assume that the

preprocessing and segmentation function reduce the image bandwidth by 100:1, i.e. a frame is now composed of 3300 words, then a frame rate of 3 per second would require a word rate of 10,000 words per second. By words we mean the following: The original image is composed of 330,000 pixels each; say, 5 bits wide. When they are preprocessed and segmented, the resulting pieces of information total about 3300 words which may be wider or narrower than 5 bits. Assuming an input rate of 10,000 words per second into the high level feature extractor and classifier, the time between words is 100 microseconds or 100,000 nanoseconds. If we assume, from Section 3, a microinstruction cycle time of 200 nanoseconds or less, the bit slice machines can execute 500 microinstructions between words. If we further assume a wide enough bit slice microprocessor, e.g. 56 bits wide such that on average about 5 microinstructions are needed per macroinstruction (ADD, SUBTRACT, etc.), then the program length can be of the order of 100 macroinstructions. If we add a sophisticated bus structure, increasing the word width to say 72 bits, then the program length might be increased to 200 macroinstructions. Going to real time opeation at 30 frames per second would reduce the possible macroinstructions by a factor of 10 unless parallel processors were employed. In the image processing case, small word widths and parallel processors may be much more suitable than one machine of substantial width. In any event, the rate with which the processor must cope is a basic rate of 10,000 words per second with a goal of achieving 100,000 words per second.

1.2 Reliability

There are really two parts to the reliability problem; they are both branches from the central problem of time on station. For example, the fire control computer built by Westinghouse for the F-16 program does not require fault tolerance modes, dynamic reconfiguration, and internal trouble shooting because the time of flight is not long. Further, the reliability

of the bit slice processor is sufficiently high that a cascade of 18 bit slice processors will exceed military reliability specifications.

On the other hand the E-3A (AWACS, Airborne Early Warning System) computers built by Westinghouse have extensive internal trouble shooting characteristics including fault isolation, dynamic reconfiguration, and fault tolerance modes because the time on station is very long compared to the F-16. Also the volume available in each case is sufficiently different. The E-3A is housed in a Boeing 707, while the F-16 is one of the smallest of the new fighters. Thus, the time on station is a critical factor in determining whether or not internal trouble shooting techniques are required.

1.3 Size

The previous remarks on reliability are entwined with those of size. There has been a general agreement among the users that approximately 1 $ft.^3$ is an appropriate size for most airborne short mission-time applications. On the other hand, the longer term missions generally can accommodate more space, which coincides with the added hardware needed for internal trouble shooting. In these cases, internal trouble shooting is necessary in order to meet reliability specifications. We shall assume in the latter case that the total volume, including trouble shooting hardware shall be no larger than 1.5 $ft.^3$. For ground based computers, size is not so important, but reliability and dynamic reconfiguration are.

1.4 Weight

There is no reason to change the airborne weight goal from that of the Smart Sensor; namely, in the area of 20 to 30 lbs. including power supply. The fault tolerant machine can be at the high end of the range and perhaps exceed it slightly since payload is not such a difficult problem. And in the ground station application, weight is only important for mobility considerations.

1.5 Probability of Detection and Recognition, and False Alarm Rate

In this work we are including high level image processing in order to handle more complex targets such as airports and camouflaged weapons and also to increase performance figures on easier targets such as weapons with small amounts of obscuration. For the Smart Sensor work, the probability of detection was 96 percent, and the false alarm rate was 1.3 per frame. It would seem reasonable to expect that the incorporation of higher level algorithms would raise the probability of detection for the simpler targets and provide at least 0.90 on the more complex targets. The false alarm rate is probably too high and should be reduced to something like one false alarm every five frames or lower. These figures, the reader must realize, are extrapolated performance figures from the previous DARPA work.[1]

1.6 Power

Our general goal for power consumption is 200-300 watts. The addition of fault tolerance hardware will increase the power requirements, but certainly anything over 400 watts is not desirable in a airborne system. Thus we expect some increase in size and power with the fault tolerance hardware. For ground stations the allowable power varies with the installation requirements.

1.7 Software Design Goals

In this section, we attempt to discuss some of the constraints within which the software should fall. For example, in Para. 1.1, we developed the idea that each program should contain 100-200 macroinstructions in order to meet the speed requirements. No doubt, the University of Maryland analysts will be running their programs in source language (composed of macroinstructions) so this number has meaning for them.

In Section 3.0, we discuss the candidate bit-slice processors and it is noted that none have trigonometric functions. The tables shown in Section 3.0 list the logic/arithmetic operations which can be performed by them. Of course, other functions may be performed, but it is not an efficient process and may require a substantial part of the allotted

100-200 macroinstructions. Another limitation is the window size in the form of the number of available registers. For example, a 4 x 4 window requires 16 registers which is available on the AMD 2901A microprocessor, but not on some of the others. Finally, random memory access to a larger capacity memory can be time consuming and we do not encourage it.

2.0 PROCESSORS

We shall be considering appropriate processors for the higher level image understanding algorithms in this section, with subsections covering architecture, microprogramming, and processing units. We shall also describe and contrast the features of several commercially available units. First we consider some definitions.

Microprocessor - A single integrated chip containing an arithmetic (ALU) logic unit; some random access memory for storing data and intermediate results; some programmable read only memory to hold instructions; and a controller for handling status flags, I/O duties, routing addresses and data, and executing instructions. They are usually manufactured in MOS technology and can execute an instruction in approximately 10 microseconds. They have a fixed instruction set, fixed architecture, and fixed word length. They can be cascaded to handle longer words but all parts of the microprocessor increase linearly in size.

Bit Slice Microprocessor - Generally, they are faster than the single chip microprocessor by at least an order of magnitude, because they are built from bipolar technology. Here the controller and arithmetic logic unit are found on two different chips, 2 to 4 bits wide. They have a flexible approach to word length, instruction set, and architecture: thus they have the capability of being specifically configured for the application and their components can be expanded independently

of each other. The designer specifies the systems instruction set by a program (microprogram) stored in a PROM (Programmable Read-Only Memory). Some significant applications of bit slice microprocessors have occurred in signal processing where algorithms lend themselves to parallel implementations.

Based upon the above capability, and in view of substantial Westinghouse experience in bit slice implementation of airborne signal processors, we recommend use of bit slice machines in the image understanding area.

2.1 Architecture

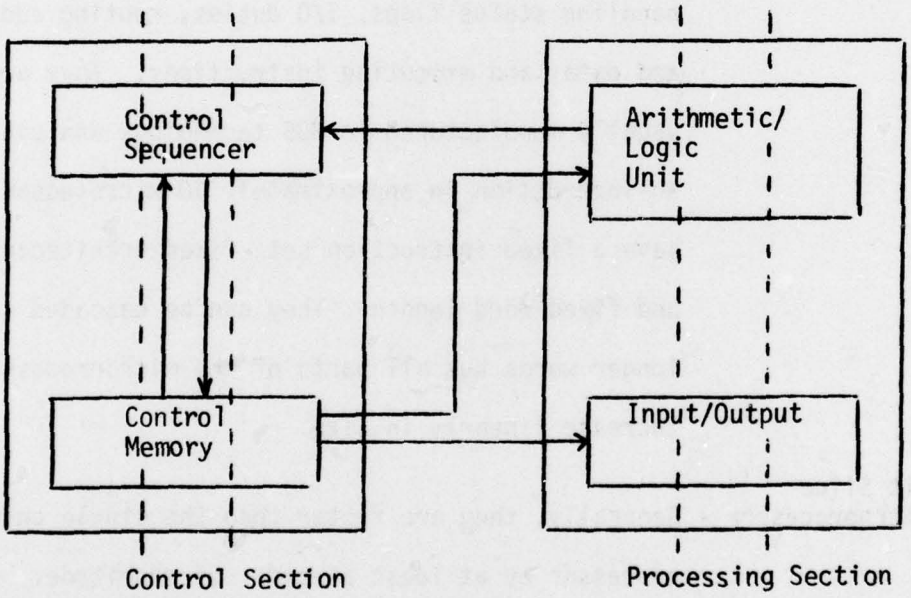


Figure 2.1. Basic Architecture - Bit Slice Microprocessor

As a starting point, we can consider a bit-slice microprocessor as composed of four sections as shown in Figure 2-1; a main memory, a control section, a processing section and an input/output section. The control and processing sections are on a minimum of two different chips; note that the dotted lines imply that the sections are composed of a number of slices. This means that in reality, the main memory, control sequencer, control memory and ALU may be built up from a number of chips (slices) and that they need not all have the same word width. The macroinstructions (ADD, SUBTRACT, etc.) and data are stored in the main memory; these are transmitted through the system by means of data and address buses. Some of the faster bit slice implementations have a rather sophisticated bus structure and control. The microprogram control memory contains the microinstructions through which the machine controls the parallel operation of the bit slice ALU's, and generates pulses timed to control the rest of the system including macroinstruction fetch from main memory. The microprogram control sequencer contains macroinstruction decode logic which maps it into a microprogram memory address and it examines all the control and status bits to determine the next microinstruction address. The macroinstruction (ADD) is executed as a series of microinstructions. Referring to Figure 2-1, the microprogram control sequencer has an address line to the microprogram control memory; the microprogram control sequencer receives status bits along control lines from the ALU and microprogram control memory. The microprogram control memory sends control signals to the processing sections. We now present an example of a macroinstruction, COMPARE, written in microcode on an imaginary 20 bit wide bit slice machine.

2.2 Microprogramming Example[2]

The COMPARE instruction is IF A = B then A = 1/2 x A, OTHERWISE GO TO NEXT INSTRUCTION. We assume that there is an instruction counter IC such

that GO TO NEXT INSTRUCTION means IC = IC + 1. And, as is the usual case, the test for A = B is performed by calculating C = A-B and comparing C with zero. A = 1/2 x A is accomplished by shifting A to the right. Then the flow chart for COMPARE is shown in Figure 2-2.

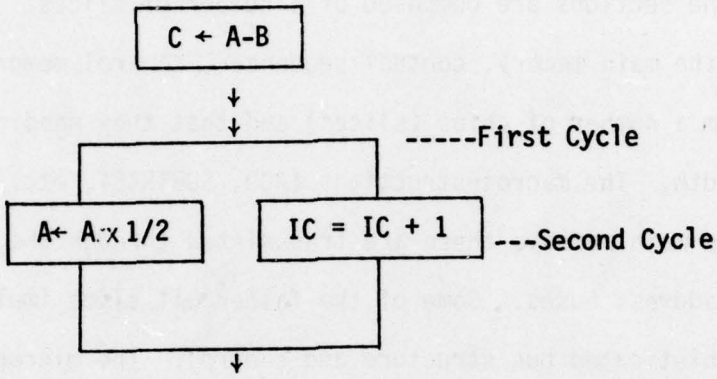


Figure 2-2. Flow Chart for COMPARE Macroinstruction

We assume a processing section configuration as shown in Figure 2-3. The two source registers A and B are represented as well as the instruction counter, IC, and the ALU. The zero detect logic issues a flag when all the bits on the data bus are zero. and the status register remembers the state

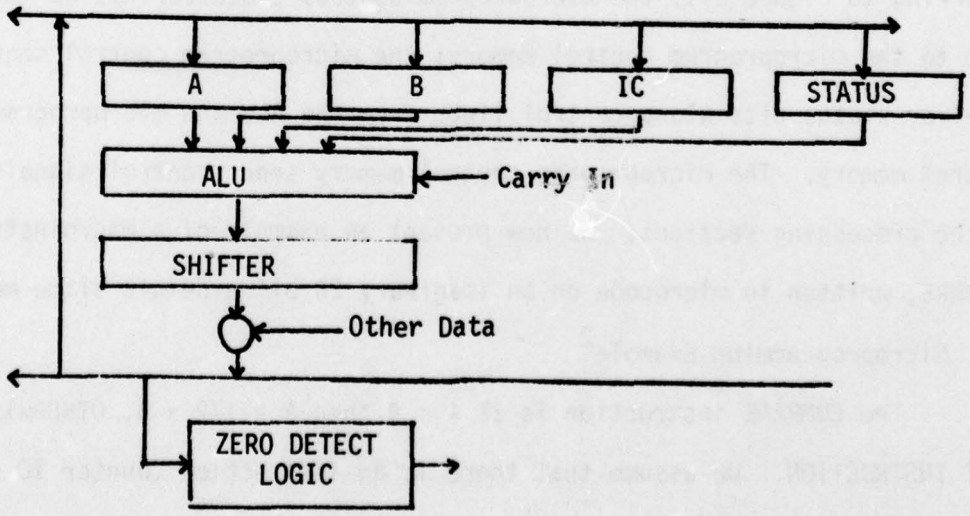
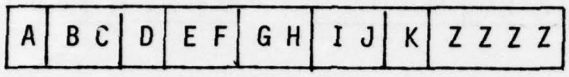


Figure 2-3. Processing Section

of the zero detect logic. Also, 1's complemented notation is used for ALU inputs, a carry-in bit is needed to convert to 2's complement as required by the ALU. We now form a microinstruction where each field corresponds to each entity in Figure 2-3.



Field A: 1 bit wide; 0, A is not gated to ALU
1, A is gated to ALU

Field BC: 2 bits wide; 00, neither B nor IC is gated to ALU
01, 1's complement of B is gated to ALU
10, B is gated to ALU
11, IC is gated to ALU

Field D: 1 bit wide; 0, addition with no carry in
1, addition with carry in

Field EF: 2 bits wide; 00, no shift of ALU output
01, shift output right one, gate to data bus
10, shift output left one, gate to data bus
11, other data gated to data bus

Field GH: 2 bits wide; 00, no destination
01, register A is data bus destination
10, register B is data bus destination
11, IC is data bus destination

Field IJ: 2 bits wide; 00, portion of address (ZZZZIJ) for next
01, microinstruction
10
11

Field K: 1 bit wide; 0, no action
1, set status register to 1 if zero detect logic detects all zeroes on data bus

Figure 2-4. Field Explanation

Then the flow chart, Figure 2-2, of indicated mathematical functions is replaced with three microinstructions which not only reflect those functions but also indicate data flow through the processing section, as shown in Figure 2-5.

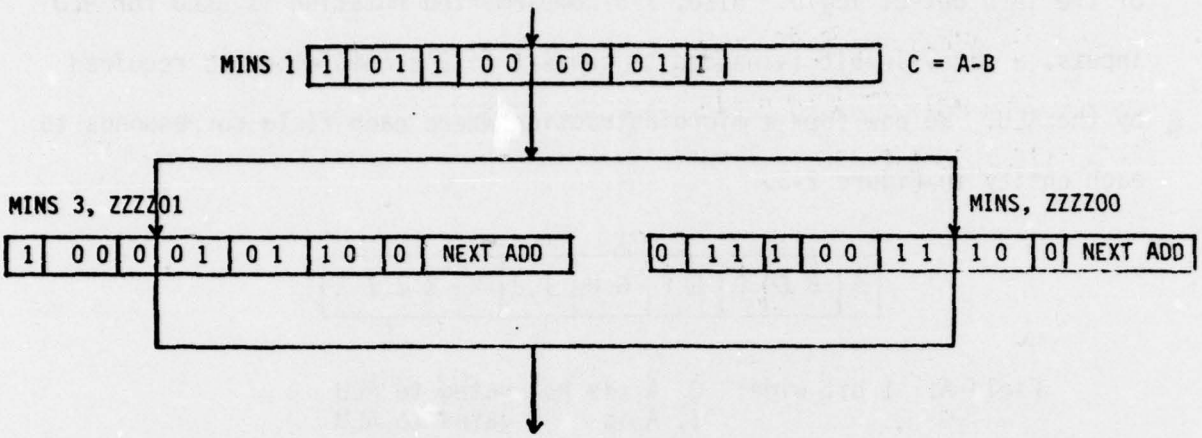


Figure 2-5. Microinstructions for COMPARE

Microinstruction 1 (MINS 1) provides that, referring to Figure 2-3,

A is gated to ALU,

B is gated to ALU in 1's complement form,

addition of A and B with carry in,

no shift of ALU output,

no destination

set 2nd address bit in field to state of status register, and

set status register to zero if zero detect logic finds all zeroes on data bus.

In MINS 1, the contents of the B register are subtracted from the A register and the result is gated to the data bus and the status register is set to zero if A = B. Since this is linked to the next instruction address, the status bit is reflected in Field IJ which determines a left or right branch.

Microinstruction 3 (MINS 3) provides that

> A is gated to ALU,
> neither B nor IC is gatedto ALU,
> addition with no carry in
> shift output right one and gate to data bus,
> register A is data bus destination,
> next microinstruction address least significant bits are specified,
> and status register takes no action.

In MINS 3, A is brought into and through ALU (A = A+0), the output is shifted right one position resulting in A = 1/2 A, and the result is deposited via the data bus to register A.

Microinstruction 2 (MINS 2) produces the following action

> A is not gated to ALU,
> IC is gated to ALU,
> addition with carry in,
> no shift of ALU output,
> IC is data bus destination,
> next microinstruction address least significant bits specified,
> and status register takes no action.

Next we examine the processing unit in some detail and contrast the characteristics of several available bit-slice processors. We shall choose from among these for implementation of a higher level image processing machine.

2.3 Processing Unit

The processing section is divided vertically, both registers and ALU, into a number of bit slices which are identical as shown in Figure 2-1. The ALU unit (Figure 2-6) typically contains an arithmetic logic unit, a set of temporary data registers which have one or two entrance and exit paths, a multiplexer which will select among a number of data sources for the ALU, a decoder for control signals, and a register to hold status bits as described in the last example.

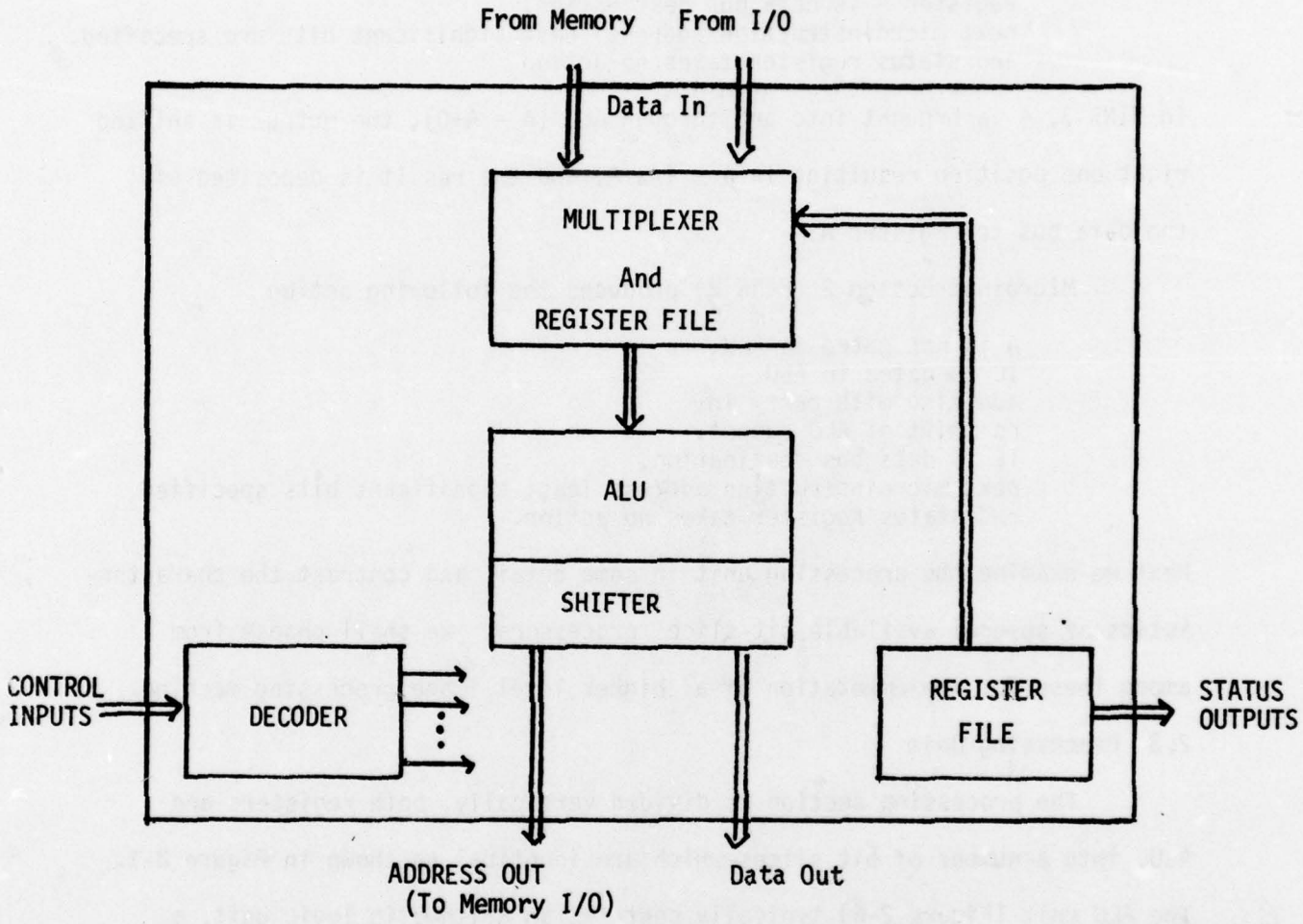


Figure 2-6. Block Diagram of ALU

Tables 2-1 and 2-2 show some of the characteristics of commercially available bit slice microprocessors; Table 2-1[3] shows the data handling characteristics, i.e., the type of registers and size, the number and kind of data ports or bus lines available, and control lines. Table 2-2[3] shows the extent of the arithmetic and logic capabilities of the ALU. The Intel 3002 is 2 bits wide while the remainder of the bit slice processors are 4 bits wide. There are some important similarities and differences among these processors which we shall note as regards their application to image processing.

TABLE 2-1. Data Handling Characteristics

		REGISTERS		DATA PORTS			
MODEL	POWER (MIN)	ACCUM.	REG. FILE	BUFFERS	INPUT	OUTPUT	BIDIRECT	CONTROL LINES
Fairchild - 9405(T^2L)	500	0	8	1	1	1	0	8
Intel-3002 (T^2L)	725	1	11	1	3	2	0	9
MMI-6701 (T^2L)	1,075	1	16	0	1	1	0	17
AMD-2901A (T^2L)	925	1	16	0	1	1	0	18
AMD-2903 (T^2L)	1,000	2	8	0	1	2	0	11
TI-SBP0400A (I^2L) SBP0401A	1,125	2	0	2	1	2	1	17
TI-SN745481 (T^2L)	1,374	1	0	1	1	0	2	17
MC-10800 (ECL)	1,155	1	16	1	1	0	2	17

TABLE 2-2. Arithmetic/Logic Functions

MODEL	OPERATIONS	OPERATIONS	BCD ADD, MULT, DIVIDE	PARITY	OTHER	DECODED STATUS
9905	C,A,O,OE	A	-	-	-	Z
3002	C,A,O,EN	A,D1,26	-	-	-	Z
6701	C,A,O,EO	A,S,D1,26	-	-	-	O.Z
2901A	C,A,O,EO,EN	A,S,D1,26	-	-	-	Z
2903	C,A,O,NA,NO, EO,EN	A,S,11,12, 2C	MULT/DIVIDE	Yes	2's Complement Sign/Magnitude Normalize	O,Z
SBP0400A SBP0401A	C,A,O,EO,EN	A,S,11,2C	-	-	-	Z
SN745481	C,A,O,NA,NO, EO,EN	A,S,11,D1, 2C	MULT/DIVIDE	-	Cyclical Redundancy Checking	O,E,AG,LG
MC-10800	C,A,O,NA,NO, EO,EN	A,S,D1,D2, I1,L2,2C	BCD ADD	Yes		O,Z

C Complement
A AND
O OR
NA NAND
NO NOR
EO excl. OR
EN excl NOR

A add
S subtract
Di Decrement by i
1i Increment by i
2C Z'complement

O overflow
E equal
Z zero
AG greater than
LG less than

The Monolithic Memories MMI-6701 is very similar in architecture to the Advanced Micro Devices 2901A processor; the register configuration, data ports, control lines, arithmetic/logic operations, and decoded status have a close correspondence. The AMD 2901A has a cycle time of 200 nanoseconds, one of the fastest built to conform to military specifications, which can be decreased further by appropriate bus structures. The processors have found ready application as control processors by cascading them to as many as 128 bits wide. Their data manipulation abilities are not as powerful (number of data ports) as others but the number of registers, 16, reduces memory manipulation and execution time.

The Fairchild 9405 finds good application in simpler processing problems where an inexpensive, simple machine is appropriate. It may not be powerful enough to handle the image processing algorithms with sufficient speed.

The AMD-2903 is an upgraded version of the AMD 2901 with a multiply/divide capability, a smaller register file, and parity checking. The arithmetic capability has been significantly enhanced and the data manipulation capability has been changed in several ways: decreasing the number of register files and increasing the number of output ports. The AMD 2901A, 2903, and MMI-6701 are primarily aimed at numerical calculations with good data manipulation characteristics. To increase manipulation ability, the designer must supply additional features such as an enhanced bus structures, microinstruction formatting, and perhaps additional temporary storages.

The Intel 3002 is two bits wide but has substantial data manipulation capability in the form of a large number of I/O ports; however, the single port register file will probably require an increased number of instructions and corresponding execution times. A fundamental consideration is the use of two chips to duplicate the word width of the other bit slice processors listed.

The Texas Instrument SBP0400/SBP0400A processor is the first ALU available in I^2L technology. Unfortunately, it turns out to be slower than any of the other listed processors, which hinders it for high throughput operations such as image processing.

The Texas Instrument T174SN481 requires memory to memory architecture because it lacks an internal register file; this is not difficult to implement but simply requires more chips. Latches are provided on the input ports so that multiple port rams can be used. An interesting feature is a multiple use ALU, i.e. several registers are designated for addressing memory so that they may be incremented while the ALU is performing some other operation. The large number of control lines facilitates data manipulation and classical signal processing techniques.

The Motorola MC 10800 is an interesting device; it is the fastest of the group being built from ECL yet its power consumption (1155 milliwatts) is less than the TI-SN745481 and comparable to the shown TI SBP0400A. It is 20 percent higher than the 2901 A which is a good competitor to it. The MC 10800, like the TI SN745481, must also be supported by an external file and has the appropriate I/O bus structure. It also provides internal parity error control and binary coded addition. At this point, we can with some margin of error, eliminate several of these devices from further consideration.

We assume at this point that the higher level image processing algorithms will require high throughput, a major emphasis on data manipulation, and a lesser emphasis on arithmetic operations such as multiply and divide. The Fairchild 9905 is probably not powerful enough for this application; there are faster machines than the IT SBP0400A with almost comparable characteristics. The MMI-6701 is directly comparable with the AMD 2901A, and our good experience with the latter points to it as the choice here. We shall keep the MC-10800

in the race because we may need the speed, and the power consumption, usually associated with ECL logic, does not seem oppressive.

3.0 APPLICATIONS OF LISP

The higher order artificial intelligence and image understanding algorithms attempt to acquire information from features already extracted from the image; one can imagine a similar situation in which a number of facts are given to an analyst and he must assemble them in a logical fashion. Thus, for the most part, the higher level algorithms will be manipulating lists of facts or symbols; this process lends itself to a language like LISP[4] (from List Processor) which is a language for manipulating list structures. As suggested earlier by Lt. Col. David Carlstrom, we shall look at the possible use of LISP as an appropriate language. Let us now spend some time working through some of the fundamentals of LISP to obtain some idea of machine requirements.

3.1 Symbolic Expressions

We define an atomic symbol as no more than thirty (30) alphanumerics which represent an entity and are not capable of being split. Also, the first character must be a capital letter, e.g. A5B63, ZQRSTWRR, M, P54321Q, etc. S-expressions are the lists and are made up of atomic symbols or other S-expressions in the following form: left parenthesis, an S-expression, a dot, an S-expression, and a right parenthesis. Some examples are: $(Z_1 \cdot Z_2)$, $(A5 \cdot (8 \cdot C))$, and $((X1 \cdot Y) \cdot Z \cdot (Y \cdot Z))$. The third expression, for example, consists of an S-expression, an atomic symbol, and an S-expression, further the first and second S-expressions consist of two atomic symbols each. Having described some symbolic definitions, let us consider five basic functions.

3.2 Basic Functions

The notation will be as follows: the function will be in lower case letters; the arguments will be grouped in square brackets and separated by a semi-colon. The function "cons" is used to build a S-expression from two smaller expressions (which, recall, may themselves be atomic symbols).

Some examples include:

cons [A1;Z3] = (A1·Z3)

cons [AAC;(BB·CD)] = (AAC·(BB·CD))

cons [cons[MN;OP];Z3] = (MN·OP)·Z3)

To obtain a divisible part of an S-expression, two different functions are employed, one to obtain the leftmost, or first subexpression, and the other to obtain the rightmost, or second subexpression. The former function is called "car" and the latter is called "cdr"; some examples follow:

car[(A·B)] = A	cdr[(A·B)] = B
car[((MN·OP)·Z3)] = (MN·OP)	cdr[((MN·OP)·Z3)] = Z3
car[Z·N16] = Z	cdr[(N16·Z)] = Z

These functions have one argument, but the functions are only defined when that argument is not an atomic symbol.

Thus far we have described three functions which are used to construct larger S-expressions or obtain divisible parts of an S-expression. Consider another type of function, one whose value is either true or false; the function "eq" is a test for equality of atomic symbols:

eq[A;A] = T

eq[A;B] = F

eq[(A·B);C] is undefined

The function "atom" is true if its argument is an atomic symbol, e.g.

atom[A] = T

atom [(A·B)] = F

atom [ABCDEFGHI1234] = T

Having described atomic symbols, S-expressions, five basic functions, and having seen something of the recursive nature of LISP, let us consider some more interesting conditional expressions. These kinds of expressions will begin to give the reader some inkling of the processing capability necessary to accommodate LISP.

3.3 Conditional Expressions

A conditional expression of the form $[c_1 \rightarrow e_1; c_2 \rightarrow e_2; c_3 \rightarrow e_3; \ldots]$ means that if c_1 is true, then the value of e_1 is the value of the entire expression. If c_1 is false, then if c_2 is true, the value of e_2 is the value of the entire expression. The c_i are searched from left to right until the first true one is found. Then the corresponding e_i is selected. If none of the c_i are true, then the value of the entire expression is undefined. There are some particular conditional expressions which we describe now, where lower case x, y, z are symbols for general arguments.

The function subst [x; y; z] gives the result of substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z. The function is defined as

```
subst [x;y;z] = [atom[z] →[eq[z;y] → x; T → z]; T → cons [subst
                    [x;y;car[z]]; subst [x;y;cdr [z]]]]
```

The expression may be put in the conditional form as:

$$\text{subst}[x;y;z] = [c_1 \rightarrow e_1; c_2 \rightarrow e_2]$$

where: $c_1 = \text{atom}[z]$

$e_1 = \text{eq}[z;y] \rightarrow x; T \rightarrow z]$

$c_2 = T$

$e_2 = \text{cons}[\text{subst}[x;y;\text{car}[z]; \text{subst}[x;y;\text{cdr}(z)]]$

and: c_1' = [z;y]

e_1' = x

c_1'' = T

e_2'' = Z .

As an example, assume that z = ((A·B)·C), x = (X·A), and y = B, then

Step 1. c_1 = atom ((A·B)·C) is false

c_2 = T is true

car [z] = (A·B)

cdr [z] = c

e_2 = cons [subst[x;y;(A·B)]; subst [x;y;C]]

and evaluating each part of cons:

Step 2. Subst [x;y;(A·B)] =

atom[(A·B)] is false

T is true

car (A·B) = A

cdr (A·B) = B

cons [subst [x;y;A]; Subst [x;y;B)] =

Subst [x;y;A]

atom[A] is true

eq [A·B] is false

T is true

Subst [x;y;A] = Z = A

Subst [x;y;B]

atom[B] is true

eq [B;B] is true

Subst [x;y;B] = x = (X.A)

then cons [subst [x;y;A]; subst [x;y;B]] = cons [A;(X·A)] = (A·(Z·A)).

Step 3. Subst [x;y;c]

atom [c] is true

eq [c;B] is false

T is true

Subst [X;y;c] = z = c

And, substituting into the original expression, we find that

$$e_2 = \text{cons}\ [(A\cdot(X\cdot A));C] = ((A\cdot(X\cdot A))\cdot C)$$

which is equivalent to substituting X = (X·A) for B in the expression z = ((A·B)·C).

Another function is equal [x;y] which is true if its two arguments are identical S-expressions and false if they are different;

equal [x;y] is defined as

equal [x;y] = [atom[x] → [atom[y] → eq[x;y]; T → F];
equal [car[x]; car[y]] → equal [cdr[x]; cdr(y)]];
T → F].

Assume x = (A·B), y = (A·B), and substituting

Step 1. atom(x) is false

car (x) = A cdr (x) = B

car (y) = A cdr (y) = B

then

equal [car[x]; car[y] → equal [cdr[x]; cdr[y]]

becomes

equal[A;A] → equal[B;B]

equal[A;A] = atom[A] → atom[A] → eq[A;A] = T

equal[B;B] = atom[B] → atom[B] → eq[B;B] = T

and equal[x;y] = T

Where the expression has the conditional structure

c_1 = atom[x]

c_2 = equal[car[x]; car[y]]

c_3 = T.

The function null[x] is used to decide if a list is exhausted; it is true if and only if the argument is NIL which is a terminator of lists i.e. c = (C·NIL) and null (NIL) = T.

If the S-expressions are regarded as lists, the "append [x;y]" and "member [x;y]" functions are useful. The "append" function puts two lists together and is defined as:

append[x;y] = [null[x] → y;T → cons[car[x]; append [cdr[x];y]]].

As an example, let x - (X1·X2), y = (X3·X4), where

c_1 = null[x]

c_2 = T

Step 1. null[x] is false

T is true

car[x] = X1 cons[X1; append[X_2,y]]

cdr[x] = X2

Step 2.

append [x;y] = append [X2;(X3·X4)]

null[x2] is false

T is true

car[X2·NIL] = X2, cdr[X2·NIL] = NIL

append [NIL;(X3·X4)] =

null(NIL) - True

append [ ] = (X3·X4)

cons[car[X2·NIL] append[NIL;(X3 X4)]] = cons (X_2;(X_3 X_4))

= (X_2 · (X_3·X_4))

Step 3.

cons[X1; (X2·(X3·X4))] = (X1 ·(X2 ·(X3·X4)))

The function "member [x;y]" is true if the S-expression x occurs among the elements of the list y and is defined as

Member[x;y] = [null[y] → F; equal [x;car[y]] → T;

T → member[x; cdr[y]]].

The function "pairlis [x;y;a]" gives the list of pairs of corresponding elements of the lists x and y, and appends them to the list a. The resultant

list of pairs, which is like a table with two columns, is called an association list. The function pairlis [x;y;a] is

```
pairlis [x;y;a]  = [null[x] → a; T → cons [cons [car [x]; car [y]];
                                    pairlis  [cdr [x]; cdr [y]; a]]].
```

As an example, consider two lists x = (X1·(X2·X3)) and y = (y1·(y2·y3)) which are to be paired and added to a list (X4·Y4)·(X5·Y5).

Step 1.

null [x] = false

T is true

car [x] = X1 cdr [x] = (X2·X3)

car [y] = Y1 cdr [y] = (Y2·Y3)

cons [cons [x1;y1]; pairlis [(x2·x3); (Y2·Y3); a]

Step 2.

cons [X1;Y1] = (X1·Y1)

pairlis [(X2·X3); (Y2·Y3);a] =

null(X2·X3) = false

T = true

car [(X2·X3)] = X2 cdr [X2·X3] = X3

car [(Y2·Y3)] = Y2 cdr [Y2·Y3] = Y3

cons [cons[X2;Y2]; pairlis [X3;Y3;a]

Step 3.

cons [X2;Y2] = (X2·Y2)

pairlis [X3;Y3;a] =

null (X3·NIL) = false

T = true

car [X3·NIL] = X3 cdr [X3·NIL] = NIL

car [Y3·NIL] = Y3 cdr [Y3·NIL] = NIL

cons [cons [X3;Y3]; pairlis [NIL;NIL;a]

Step 4.

cons [X3;Y3] = (X3·Y3)

pairlis [NIL;NIL;a] =

null (NIL) = True

pairlis [NIL;NIL;a] = a = (X4·Y4)·(X5·Y5)

and substituting back,

cons [cons[X3;Y3]; pairlis [NIL;NIL;a] =

cons [(X3·Y3); (X4·Y4) ·(X5·Y5)]

= (X3 Y3) · (X4 Y4) · (X5·Y5) = pairlis [X_3;Y3;a]

Similarly

cons [con[X2;Y2]; pairlis[X3;Y3;a]]

= (X2·Y2)·(X3·Y3)·(X4·Y4)·(X5·Y5)

= pairlis [(X2·X3);(Y2·Y3);a]

Finally

cons [(X1·Y1); pairlis [(X2·X3); (Y2·Y3);a]

= (X1·Y1)·(X2·Y2)·(X3·Y3)·(X4·Y4)·(X5·Y5) = Pairlis [X;Y;a]

We may then search the association list formed by pairlis and obtain the first pair whose first term is, e.g.X = X3, by means of the assoc function defined as

assoc [X;a] =[equal [caar[a];X] → car[a]; T → assoc [X;cdr[a]]]

where: caar = car[car[ ]]

There are other interesting functions in the LISP repertoire, but it does not serve our purposes to analyze them in a quarterly report. The functions described thus far do serve to give some indication of the basic LISP structure and raises some issues for further investigation which we discuss now.

3.4 Compiling, Interpreting, Machine Language and Subroutines

It is worthwhile to define the words "source program", "assembly program", "compiler", "interpreter," and "machine language" before we proceed very far into the discussion. These words are used frequently in computer technology. An assembly program translates from symbolic instructions, source program, into the language of a machine. The statements in an assembly language are, generally, one to one with the machine language to which they translate. Unlike a machine language, an assembly language allows the programmer to use symbols with mnemonic significance. A compiler is a program which translates from a source language into machine (or assembly) language. An interpreter executes a source language program by examining the source language and performing the specified algorithms. This is in contrast to a compiler which translates the source language program into machine (the object) language for a subsequent execution. As the title to this section suggests, we have a choice of operational modes, namely to compile the source program, interpret it, or write it directly in machine language with subroutines for some of the functions.

It takes about 20 times longer running time to include an interpreter in the operational machine. Of course, the memory saving is large. But with larger, cheaper, semi-conductor memories and running time at a premium, it seems reasonable to lean towards the idea of microprogramming the subroutines forming a macroinstruction of LISP.

There are several other issues also. We note from some of the preceding examples that the functions are recursive; there are a number of steps involved with storage required for intermediate results; there is an order in evaluating an expression, i.e. certain inner expressions are evaluated first; and there is a substantial amount of substitution. This immediately seems to suggest several levels of memory including a fast memory for intermediate

results; it also suggests some sort of two-level execution in which, for example, the appropriate part of a conditional statement is selected, i.e. the c_i (conditions) are extracted ahead of time and then executed in parallel when the ALU (Arithmetic Logic Unit) directs its attention to it. The left-most true condition is then enacted. Also, if there are a significant number of members in a list, the ALU does not have to be as wide as the list if the list length is characterized somehow. In other words, a list stored in memory as (A·(B·(C·(D·(E·NIL))))) = (A B C D E) could be characterized, when brought into the ALU, as (A·(G)·E) and car and cdr can be obtained without ever knowing what G is. This sort of pre-processing could significantly cut down execution time, reduce required bus, ALU, and memory widths at the price of several levels of processing and achieve a size and speed compatible with airborne requirements. Another related issue is computer memory structures compatible with lists which we examine now.

3.5 List Structures

LISP utilizes a tree structure for storing lists inside the computer instead of sequences of binary coded characters. A computer word is shown in Figure 3-1 as a rectangle divided into two sections called the address and decrement, each of which is a 15 bit field of the word. A pointer to a computer

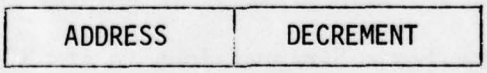


Figure 3-1. Computer Word

word is defined as the 15 bit complement of the address. We represent the condition where the decrement of word A is a pointer to word B as shown in Figure 3-2.

Figure 3-2. Word Pointer

Figure 3-2 also shows the condition where word B contains a pointer to the atomic symbol ABCDE in its address. Consider now some memory structures for several S-expressions.

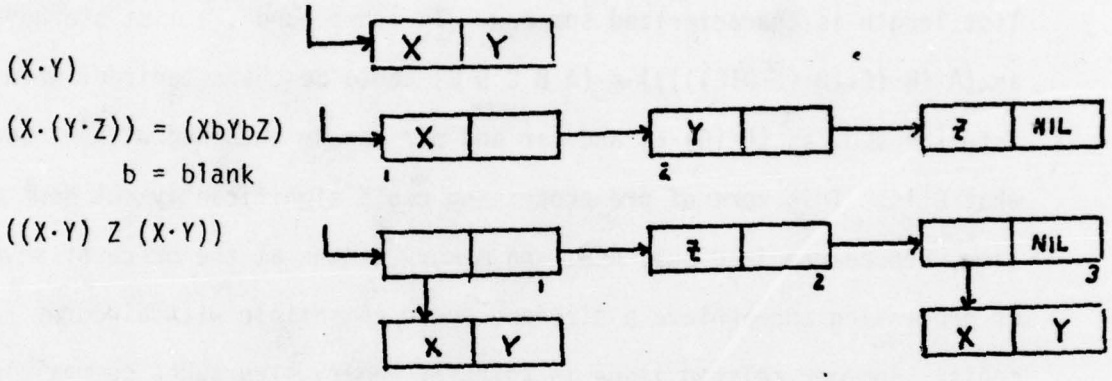


Figure 3-3. Typical Memory Structures

In Figure 3-3, the first structure represents the basic S-expression which is composed of a left parenthesis, an atomic symbol, a dot, an atomic symbol, and a right parenthesis. A NIL pointer is not necessary in the memory structure. The second structure has an X pointer in the address of word 1 but use the decrement as a pointer to the rest of the expression. Similar comments apply to word 2. Word 3 has a NIL pointer in its decrement. The third structure has two basic S-expressions in it which can both be put in a single word. The only problem remaining is to point to them properly. Word 1 points to the first (X·Y) and its decrement points to the rest of the expression. Word 2 points to the Z term and the remainder of the expression. Word 3 points to the final (X·Y) and the NIL termination. Another example is the expression (X·(Y·(Z·X))·(Z·X)), whose memory structure is shown in Figure 3-4.

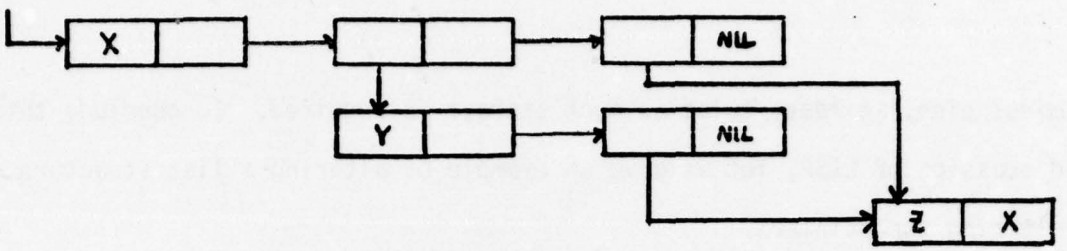


Figure 3-4. Memory Structure for (X·(Y·(Z·X))·(Z·X))

The expression may be written as (X·G·(Z·X)) where the first line of words represent X and (Z·X) and the second line represents G. A simpler way to form the structure is seen in Figure 3-5, which is similar to the third structure of Figure 3-3.

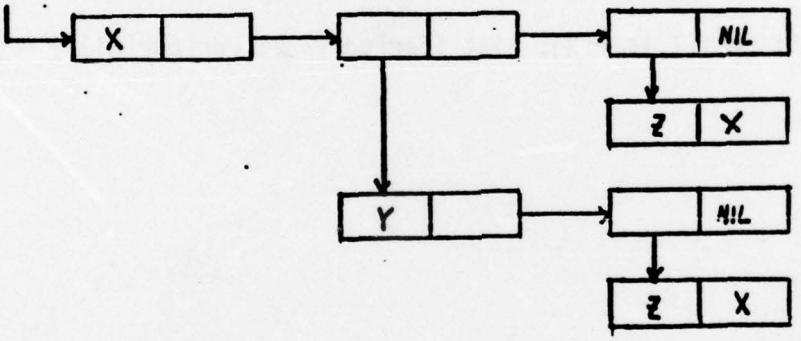


Figure 3-5. An Alternate Structure

The advantages and disadvantages of such a memory structure are stated in quite general terms. As is the usual case, the particular application will provide the appropriate weights to the plusses and minuses. On the plus side, it has been suggested that the size and number of expressions which the program must handle cannot be known in advance. Therefore, it is difficult to arrange blocks of storage of fixed length to contain them. In point of fact, this is done all the time. A second advantage is the use of pointers which faciiitate a change in list structures by changing pointers. On the

minus side, at least twice as much storage is required. To conclude this discussion of LISP, let us give an example of altering a list structure and altering the pointers.

Consider a list structure of the form ((ABC)(DEF)..., (XYZ)) which we want to alter to (A(BC))(DEF)),...-(X(YZ))). The function LIST is used in the sense that LIST [X] provides a list of the arguments of X. For example, if X1 = A, D, G, ... X, X2 = B, E, H, ... Y, and X3 = C, FI, ... Z then LIST [X1; X3; X3] = ((ABC) (DEF), ... (XYZ)). The expression X1 can be written as car[X] and LIST [car[X]] = A, D, G, ... X. Similarly, cdr[ABC] = BC so that car[cdr[X]] = B, and cdr[cdr[X]] = C. Then the new LIST (A(BC)...) may be formed by setting LIST [car[X]; list [car[cdr[X]]; cdr[cdr[X]]]] = grp[X].

4.0 ALGORITHM DEVELOPMENT

The purpose of this section is to describe the relaxation process developed by the University of Maryland and ways of implementing it in an airborne digital machine.

4.1 Relaxation (discrete case)[5]

For the first quarterly report we shall confine ourselves to the discrete Relaxation case which will be superseded in the future by the probabilistic case. However, it serves as a good introduction to the algorithm and hardware implementation. Relaxation is essentially an iterative technique where the relationships between objects are used to classify them; specific image characteristics (objects) are used to classify the image figures. For example, the objects could be line segments detected in the image. If there were four of them and they were at right angles, one might conclude that they formed some sort of a rectangular figure. Objects can also be other image characteristics such as blobs, straight lines, or junctures which, by themselves, do not have much meaning. But considered together, the classification of the figure becomes apparent. We examine the relationships for consistency, i.e. if the objects form a particular figure (rectangle), they must have a certain relationship to each other for each part of the figure. Further, inconsistent relationships must be rejected. In a deeper cut through the problem, we may perform the iteration to find a consistent relationship by discarding inconsistent relationships.

To show how this is done let us return to our previous example. Suppose the four objects have several possible relationships between pairs, and we are considering the relationships at the pair-wise level only. One way to iterate is to assume a certain classification for object number 1 and cycle through the relationship between object number 1 and each of the other objects in parallel. If the classification of object number 1 is inconsistent with one of the other

objects, the classification for object number 1 is rejected and the next classification is tried. Clearly, the analyst can end up with a set of consistent classifications, none of which dominates. This is the shortcoming of the discrete case and is handled in the probabilistic approach. The next item of interest is how the relationships are examined for consistency, as outlined in the Maryland paper.[5]

Assume there are three objects a_1, a_2, and a_3 and their possible classification can be λ or μ. More specifically, a kind of graph can be formed as shown in Figure 4-1. The dots show that the objects can all be represented as a λ or a μ. If it were

	a_1	a_2	a_3
λ	.	.	.
μ	.	.	.

Figure 4-1. Graph Form of Relaxation

not possible, e.g. to represent a_3 as a μ, there would be no dot at the (μ, a_3) position. Suppose, further, that the following arbitrary set of relationships exist between the objects.

$$\Lambda = \Lambda_1 = \Lambda_2 = \Lambda_3 = \{\lambda, \mu\} \quad (1)$$

$$\Lambda_{12} = \Lambda_{23} = \{(\lambda,\lambda), (\mu,\mu)\} \quad (2)$$

$$\Lambda_{13} = \{(\lambda,\mu), (\mu,\lambda)\} \quad (3)$$

(1) states that the objects a_1, a_2, a_3 can be represented as either λ or μ, i.e. $\{\lambda,\mu\}$. (2) states that the relationship between a_1 and a_2 is the same as that between a_2 and a_3 and can be characterized as λ for each or μ for each. (3) states that the relationship between object a_1 and a_3 can be stated as either λ for a_1 and μ for a_2 or μ for a_1 and λ for a_2. Then

these relationships can be drawn as arcs on the graph as shown in Figure 4-2.

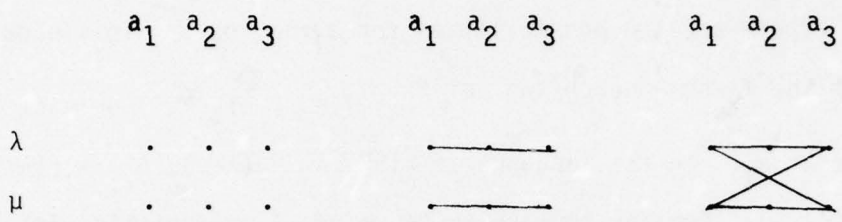


Figure 4-2a. Relationship (1)

Figure 4-2b. Relationship (1) + (2)

Figure 4-2c. Relationship (1) + (2) + (3)

Now, we see from Figure 4-2c, that there is an arc between each of the objects which symbolizes the idea that there is a consistent relationship among them. However, if we trace our way around the graph we find that $a_1 = \lambda$, $a_2 = \lambda$, and $a_3 = \lambda$ but to return to a_1 means that $a_1 = \mu$ - a contradiction. On the other hand, the graph of Figure 4-3 represents a case when there are two consistent and possible interpretations of the set of relations. The two consistent classifications from Figure 3 are (λ, λ, μ) and (μ, μ, λ) for objects a_1, a_2, a_3 respectively. Next we consider hardware implementation.

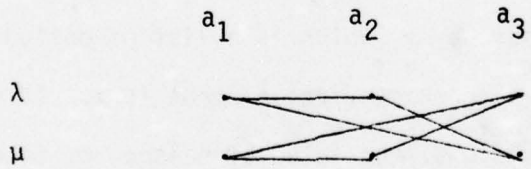


Figure 4-3. Two Consistent and Possible Situations

To form the graphs on a digital machine, we assume a_1 is classified as λ. We cycle classifications for a_2 and a_3 against it by matching λ's. So for a_2, we obtain (λ,λ) and for a_3 we obtain (λ,μ). Then for a_1, a_2, a_3, we obtain (λ, λ, μ). Similarly, assuming $a_1 = \mu$, we obtain (μ, μ, λ). These are the same two consistent classifications shown in the graph of Figure 3.

Since we are manipulating lists of symbols rather than lists of numbers, a natural computer language for this problem is LISP.[2] Referring to LISP, we note that some defined functions are directly applicable to the problem. First of all, there are two possibilities for a_1, λ, or μ. This should be compared with the first of each two of Λ_{12}, i.e. λ of (λ,λ) or μ of (μ,μ) which represents a_1. In the language of LISP, $\Lambda_1 = (\lambda\cdot\mu)$, $\Lambda_{12} = [(\lambda\cdot\lambda)\cdot(\mu\cdot\mu)]$, $\Lambda_{13} = [(\lambda\cdot\mu)\cdot(\mu\cdot\lambda)]$. Further to make the form of Λ_1 compatible with that of Λ_{12} and not change the meaning of Λ_1, we let $\Lambda_1 = [(\lambda\cdot\mu)\cdot(\lambda\cdot\mu)]$. Then we could employ the pairlis and equal functions sequentially. The function pairlis [x;y;a] gives this list of pairs of corresponding elements of the lists x and y, and appends them to the list a. As an example, let x = (X1·(X2·X3)) and y = (Y1·(Y2·Y3)) which are to be paired and added to a list (X4·Y4)·(X5·Y5), then pairlis [x;y;a] = (X1·Y1)·(X2·Y2)·(X3·Y3)·(X4·Y4)·(Y5·Y5). Then pairlis $[A_1;A_{12};1]$ = $(\lambda\cdot\lambda)\cdot(\mu\cdot\lambda)\cdot(\lambda\cdot\mu)\cdot(\mu\cdot\mu)$ and equal $(\lambda\cdot\lambda)$ = True; we obtain λ for a_2 from the second pair, assuming we remembered a_2's position in that pair. Simultaneously, pairlis $[A_1;A_{13};1]$ is computed to obtain the classification for a_3. A more direct approach in LISP is the "SASSOC" function which has the following definition:

sassoc [x;y;μ]: searches y, which is a list of dotted pairs for a pair whose first element is x. If such a pair is found, the value of sassoc, μ, is this pair.

```
sassoc [x;y;μ] = [null[y]   → μ [ ]; eq [caar [y]; x] → car [y];
                  T → sassoc [x; cdr [y]; μ]]
```

Applying sassoc,

$\Lambda_1 = [\lambda,\mu]$, car $[\Lambda_1] = \lambda = x$

$y = \Lambda_{12} = [(\lambda\cdot\lambda)\cdot(\mu\cdot\mu)]$

then,

sassoc $[\lambda;((\lambda\cdot\lambda)\cdot(\mu\cdot\mu));\mu]$ =

Step 1.

null [y] is false

caar $[(\lambda\cdot\lambda)\cdot(\mu\cdot\mu)]$ = car $[\lambda\cdot\lambda]$ = λ

eq [carr [y];x] = eq $[\lambda;\lambda]$ = T

sassoc = car [y] = $(\lambda\cdot\lambda)$.

The pair has been found, the second atomic symbol of the S-expression is the classification for a_2, namely λ. Now, we repeat sassoc for Λ_{13} to find a consistent classification for a_3.

$\Lambda_1 = [\lambda,\mu]$, car $[\Lambda_1] = \lambda = x$

$\Lambda_{13} = [(\lambda\cdot\mu)\cdot(\mu\cdot\lambda)] = y$

then

sassoc = $[\lambda;((\lambda\cdot\mu)\cdot(\mu\cdot\mu));\mu]$ =

Step 1.

null [y] is false

caar $[(\lambda\cdot\mu)\cdot(\mu\cdot\mu)]$ = car $[\lambda\cdot\mu] = \lambda$

eg [caar [y];x] = eq $[\lambda\cdot\lambda]$ = T

$\therefore$

sassoc = car (y) - $(\lambda\cdot\mu)$

and the classification for a_3 is the second atomic symbol in the S-expression, i.e., μ for a_3. And the classification becomes (λ,λ,μ) for (a_1, a_2, a_3). We would then repeat the procedure above where a_1 starts with μ, and we obtain (μ,μ,λ) for (a_1,a_2,a_3). In summary, we have shown that there are at least two LISP structures which produce the consistent classification lists for objects a_1, a_2, a_3 as also shown in the graph of Figure 3.

In terms of bit slice implementation, we might assign a processor to each object. The width of each processor would be the width of the classification word. It is worth pointing out that, since each processor would be doing the same thing, it is possible to have one controller for all three CPU's. This reduction in hardware is not possible with microprocessors which are not bit slice.

5.0 BIT SLICE PROCESSORS

Westinghouse has substantial experience in building avionics processors (high speed, small size) from bit slice components; we have built process controllers, general purpose machines, signal processors, and processors for satellite applications where reliability must be extremely high. The general purpose avionics processor (a member of the Westinghouse Milli-EP family) described in this section is an example of a bit slice CPU and bus controller; it has the following characteristics:

Volume	$< 1\ ft^3$
Throughput	400,000 operations/sec.
Memory	128,000 words, 16 bits each
Mean Time Before Failure	2027 hours
Instruction Set	Proposed Air Force Standard
General Registers	16
Indirect Addressing	Unlimited
Fast Floating Point	32 or 48 bit
Bus Structure	Microprogrammed
CPU	Microprogrammed
Bit Operation	1, 8, 16, 32, or 48
Power	600 watts

A functional block diagram of the computer is shown in Figure 5-1.

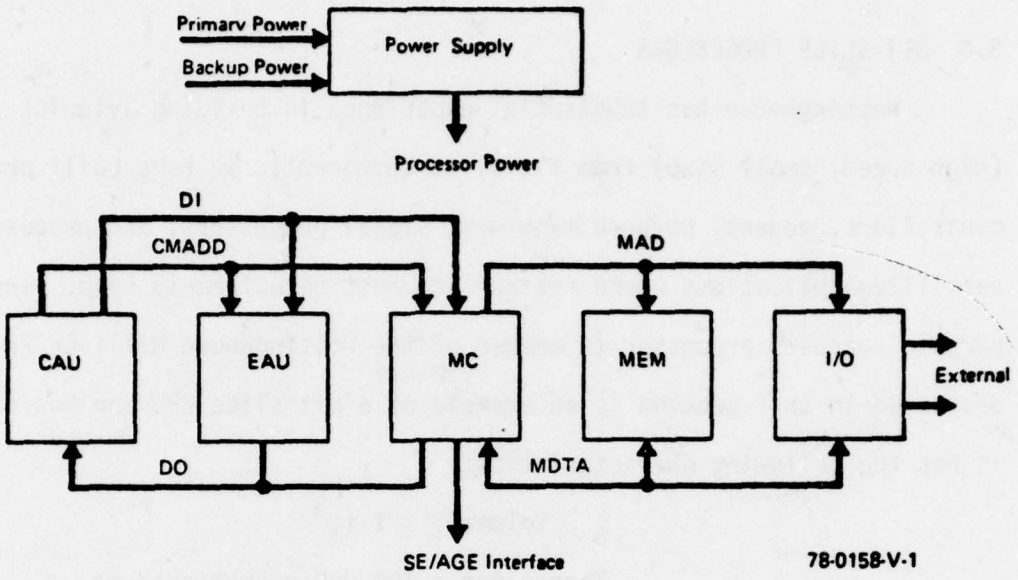


Figure 5-1. Computer Functional Block Diagram

5.1 Functional and Performance Characteristics

5.1.1 System Architecture

The architecture of the avionics processor AN/AYK-15A is shown in Figure 5-2.

The processor portion is composed of the CAU (Central Arithmetic Unit), the EAU (Extended Arithmetic Unit), and the MC (Memory Controller). These modules are functionally partitioned for ease of built-in test.

The I/O portion is comprised of standard military interfaces, interrupts, discrete inputs and outputs, interval timers, watchdog timer, real time clock, and memory protection circuitry. The memory system consisting of 128K of core memory expandable to 256k, 4k of bootstrap ROM, and 2kX32 main memory ROM is interfaced directly to the I/O bus. The support equipment interface consists of a serial computer control port. The computer control port is the means by which operator control and maintenance of the computer is achieved.

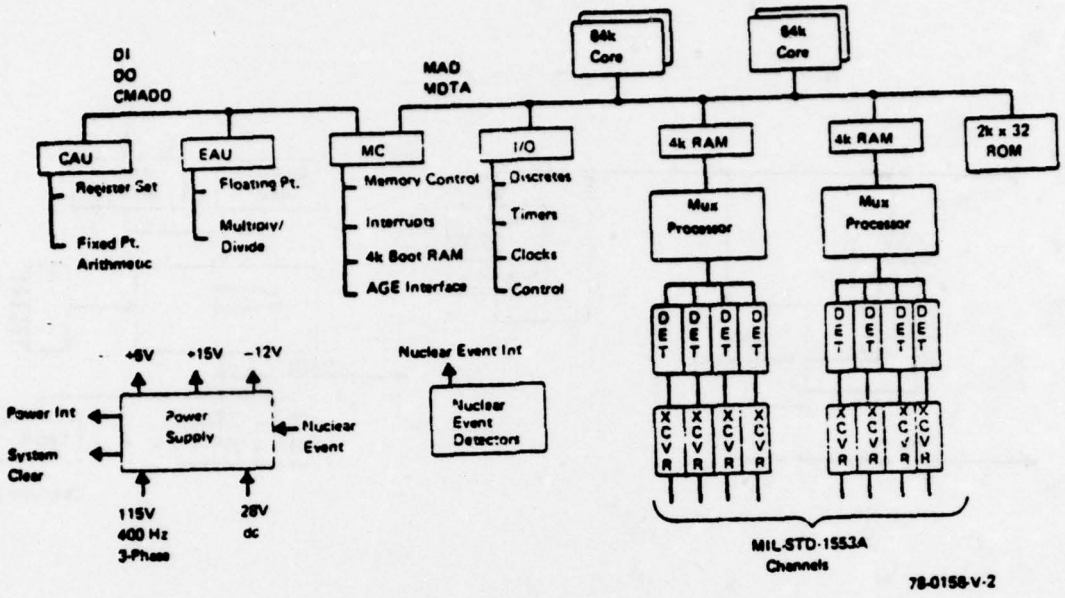


Figure 5-2. General Purpose Avionics Processor

5.1.2 Central Arithmetic Unit, CAU

The CAU performs all the fixed point arithmetic operations necessary to implement the instruction set. The CAU module is structured around the AM-2901A bipolar LSI microprocessor as shown in Figure 5-3. The microprocessor is 16-bits wide and contains 16 general purpose registers for arithmetic and indexing. Two registers (RSAV) latch the general-purpose register numbers during an instruction fetch. Their outputs are multiplexed to give register number information to the AM-2901A. A four-bit counter (MDCT) catches the shift count for shift instructions. It is also used as a sequence counter and to generate the data bit mask (through the BITMSK decoder) for data bit manipulation.

The AM-2901A is integrated with the other system elements by way of two 16-bit buses, DI and DO. The DI bus is the data interface to the microprocessor. Information from the MC, I/O, and EAU are routed to the microprocessor on this bus.

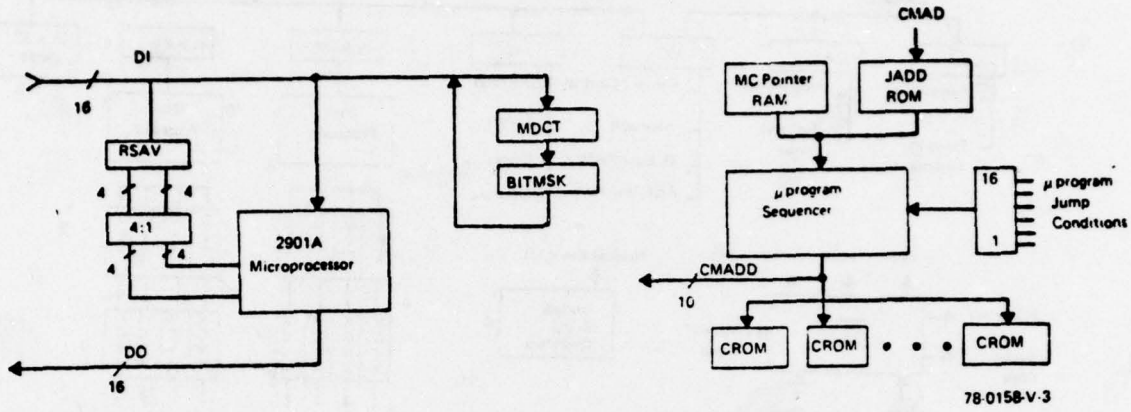


Figure 5-3. CAU Organization

The DO bus is the output data bus from the microprocessor. Results of the microprocessor operations may be routed on this bus to the MC, I/O, or EAU registers.

The control structure for the processor is implemented with a microprogram control store as illustrated in Figure 5-3. Schottky LSI microprogram sequences are used to control the sequencing of addresses to the control ROMs (CROMs). The sequences provide both conditional branch capability, as well as a "push-down stack" for microprogram subroutining.

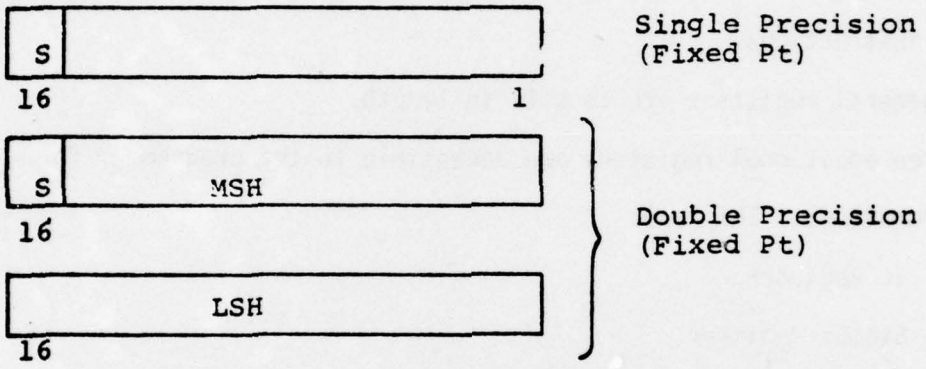
Address sources for the microprogram sequencer may come from one of three Pointer ROMs. These ROMs are used to translate system states into starting addresses for microprogram control routines. The MC pointer ROM translates the order type field into a starting address for instruction execution. In a like manner, the JADD ROM is an address source for the microprogram sequencer.

Microprogram address modification is provided by means of a 16 to 1 conditional branch multiplexer. Appropriate system flags are selected by this multiplexer for microprogram testing.

The microprogram address (CMADD) is supplied to the MC, EAU, and I/O to control sequencing.

5.1.2.1 Data Formats

The CAU module performs fixed-point arithmetic on 16-bit data (single precision) and on 32-bit data (double precision). The data formats are shown below:



In addition, the CAU performs operations on 8-bit data (byte operations) and single-bit data (bit operations).

All fixed-point data operations are performed using two's complement integer arithmetic (binary point at the extreme right end of the data).

5.1.2.2 CAU Registers

The CAU contains a set of 16 general registers for use by the programmer for arithmetic operations and address modification. Certain registers have implied usage as follows:

- Registers R1, R2, ..., R15 may be used as index registers for those instructions having the RX field.
- The registers may be partitioned as 16 single-precision (16-bit) accumulators, 8 double-precision (32-bit) or floating-point accumulators, 4 extended-precision floating point accumulators, or any combination of the above.

- Four registers, R4, R5, R6, R7 may be used as base registers for instructions having the Base Relative Address Mode.
- For instructions having the Base Relative Addressing Mode, R0 is the accumulator for double-precision and floating-point operations, and R2 is the accumulator for single-precision and integer operations.
- R15 is the implicit stack pointer for the Push and Pop Multiple instructions.

All the general registers are 16 bits in length.

Three additional registers are accessible to the programmer for certain operations. They are:

a. IC Register

b. Status Register

c. Interrupt Mask Register

The IC is 18 bits long. The other two registers are 16 bits.

The IC register directly addresses 256K of memory to point to the next instruction to be executed. It is incremented by hardware during the instruction fetch machine cycle. The IC register may be loaded by executing any of the following instruction types:

a. Jump Instruction

b. Subroutine Jump Instruction

c. Load PSW Instruction

d. Return from Interrupt Instruction

To provide maximum software flexibility to accommodate the 18-bit IC, instructions are provided in two formats: normal 16-bit address fields (which leave the upper two IC bits unchanged) and long 18-bit address fields for jumping to any of 256k locations (refer to Section 5.1.2.3). Upon interrupt, the upper two bits of the IC are automatically saved in the status register.

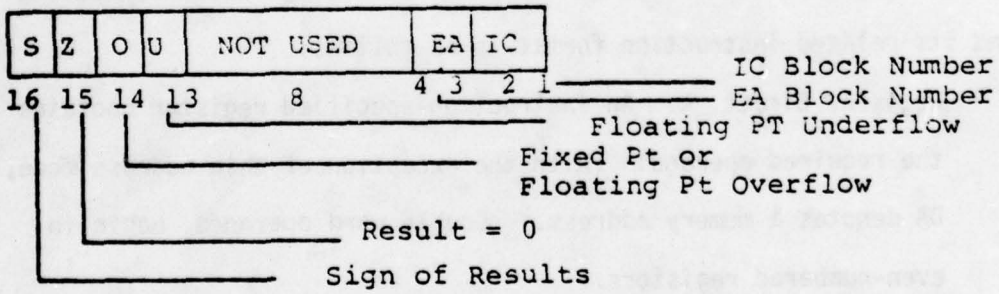
Return from interrupt automatically restores these bits along with the rest of the IC. This implementation provides symmetry for software and compiler operation, providing maximum efficiency.

A similar 2-bit extension is associated with the operand address. These extensions may be loaded via the Load Block I/O command and read via the Read Block I/O command. A Move instruction is provided to allow blocks of data to be efficiently moved anywhere in 256k of memory.

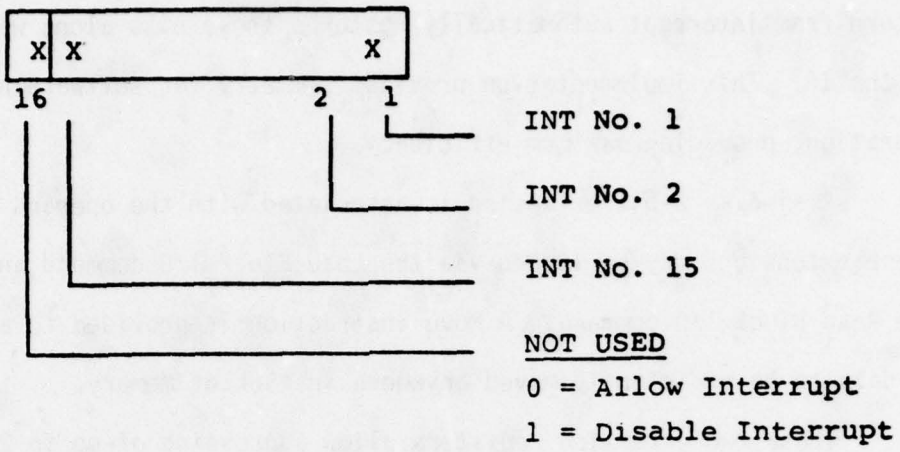
These two extension registers allow addressing of up to 256k memory half-words for both instructions and operands. If instructions are allocated to block 2 and operands to block 1, no block switching would be required during a program running on the basic 128k processor.

The status register reflects the current arithmetic status of the processor. Its format is shown below. The status word is automatically updated by hardware after execution of every arithmetic instruction. Additionally, it is loaded by the LPSW instruction.

STATUS REGISTER



The interrupt mask is a 16-bit register in the priority interrupt system used to individually mask interrupts. It is loaded when executing either the LPSW instruction on the OUT (interrupt mask) instruction. Its format is shown below.

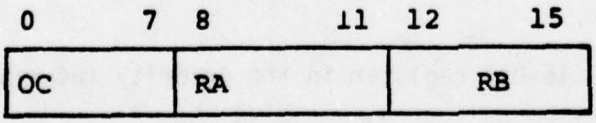


5.1.2.3 Addressing Modes

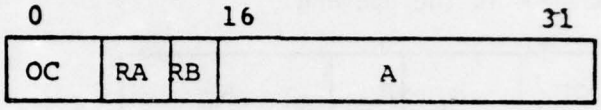
Although the memory system is defined in terms of 32-bit words, the system can efficiently handle 16-bit half-words. These half-words can be directly addressed by all instructions using 16-bit data, and 16-bit instructions and data can be packed 2 half-words to a full-word with no penalty to memory space or throughput. Instructions for loading and storing bytes can directly address the upper or lower byte of a half-word.

The CAU provides 10 modes of operand addressing. Each addressing mode and its related instruction format is as follows:

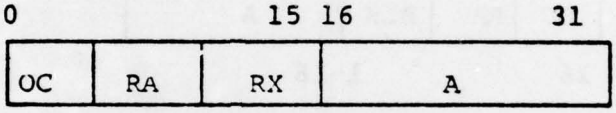
- Register Direct, R: An instruction-specified register contains the required operand. (With the exception of this address mode, DA denotes a memory address.) Double word operands begin in even-numbered registers.



- Memory Direct, D: An instruction-specified memory address contains the required operand:



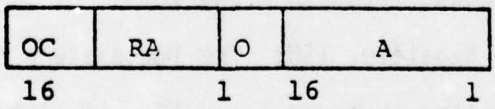
- Memory Direct-Index, DX: The memory address of the required operand is specified by the sum of the content of an index register and the instruction address field. Registers R1, R2,..., R15 may be specified for indexing.



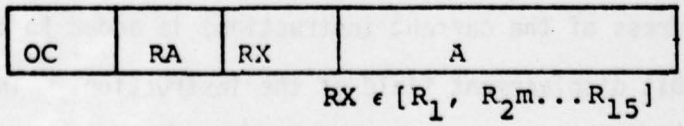
RX = 0 (Nonindexed)

RX ≠ 0 (Indexed)

- Memory Indirect, I: An instruction-specified memory address contains the address of the required operand. The address is a 32-bit word containing an 18-bit address plus a bit which specifies whether the instruction should go indirect another level.

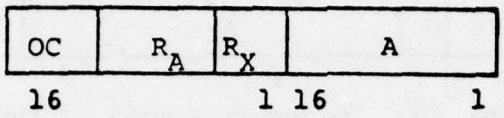


- Memory Indirect with Preindexing, IX: The sum of the content of a specified index register and the instruction address field is the address of the address of the required operand. Registers R1, R2, ..., R15 may be specified for preindexing. The indirect address is in the same format as for Memory Indirect.

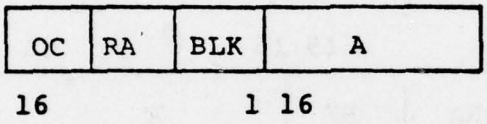


RX ϵ [R_1, R_2m...R_{15}]

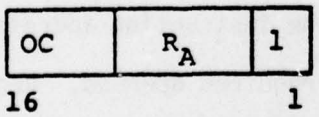
- Immediate Long Indexable: The contents of R_X when added to the address field, A, is the operand if $R_X \in [R_1,...R_{15}]$. If $R_X = 0$, then A is the operand.



- Long Address: The R_B field contains an extension of the address field (BLK). These instructions provide the capability to jump to any word in memory or move data from any word in memory.

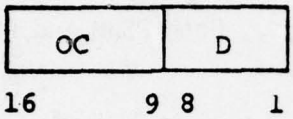


- Immediate Short, IS: The required (4-bit) operand is contained within the 16-bit instructions. There will be two methods of Immediate Short addressing; one which interprets the content of the immediate field as positive data and one which interprets the content of immediate field as negative data.
- Immediate Short Positive, ISP: The immediate operand is treated as a positive integer between 1 and 16.
- Immediate Short Negative, ISN: The immediate operand is treated as a negative integer between 1 and 16. Its internal form will be a two's complement, sign-extended 16-bit number

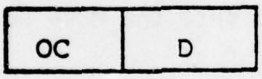


- IC-Relative, ICR: This address mode is used for 16-bit branch instructions. The content of the instruction counter (i.e., the address of the current instruction) is added to the sign extended 8-bit displacement field of the instruction. The sum points to the memory address to which control may be transferred if a branch

is executed. This mode allows addressing within a memory region of -128_{10} to 127_{10} words relative to the address pointed to be the instruction counter.



- Base Realtive, B: The content of an instruction-specified base register is added to the 8-bit displacement field of the (16-bit) instruction. The displacement field is taken to be a positive number between 0 and 255_{10}. The sum points to the memory address of the required operand. This mode allows addressing within a memory region of 256_{10} words beginning at the address pointed to by the base register.



- Stack Addressing: The CAU provides for a register/memory stack mechanism. Two instruction formats are permitted. The first format allows any of the 16 general registers to be designated as the stack pointer, while the second format uses R15 as the implied stack pointer.

  The stack is formed in memory by execution of appropriate micro-program routines to implement a "last in first out" (LIFO) algorithm. "Stacking" will proceed by loading data into successively larger memory addresses. Stack overflow will be detected as a memory protect violation when the stack area advances into protected memory.

Four instructions are provided for stack manipulation. The SJS and URTS instructions provide for subroutine stack linkage allowing any register to be designated as the stack pointer. PSHM and POPM provide register-to-stack capability for stacking and unstacking 1 to 16 general registers. Both PSHM and POPM use R15 as an implied stack pointer.

5.1.3 Memory Controller

The memory control (MC) contains the control necessary to interface the CAU with the memory system (Figure 5-4). The MC is implemented with Schottky logic. The MAD bus provides address information for memory and I/O operations. The DI and DO buses comprise the data exchange buses for communication with the CAU. Memory activity is initiated by microprogram signals which direct control logic to begin a memory cycle. When the appropriate memory is ready, the control logic sequences its data onto the MDRA bus (Read) for transfer to the MC.

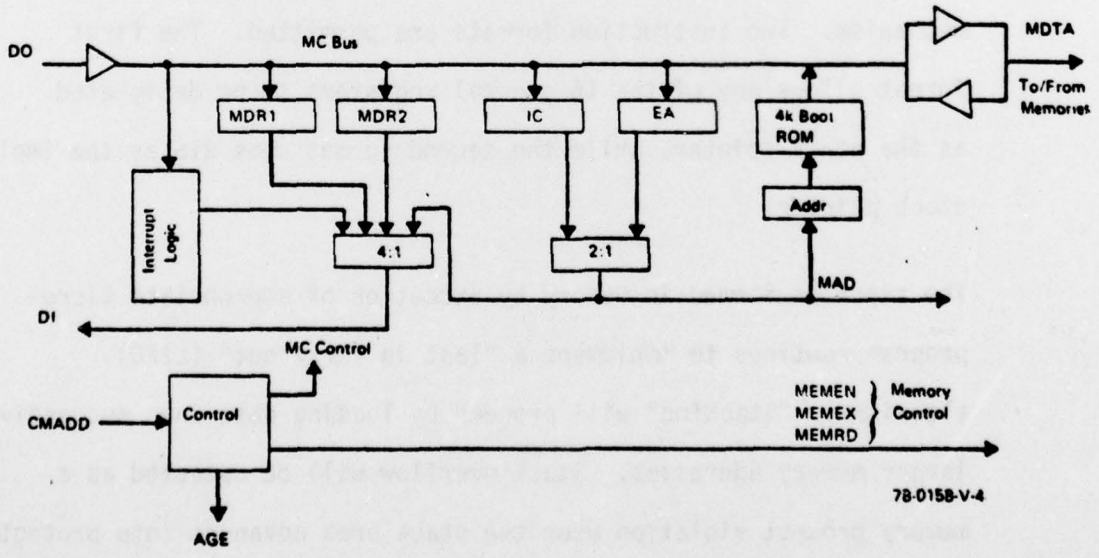


Figure 5-4. MC Architecture

Two 16-bit registers (MDR 1&2) are provided to hold data words from the memory. The IC counter supplies instruction addresses to the memory, while the EA counter supplies data addresses. All of these registers can be loaded from the CAU via the DO bus or read by the CAU via the CI bus.

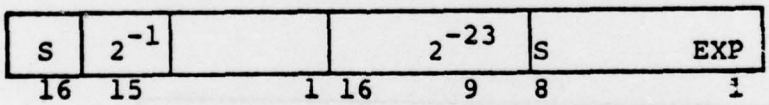
The priority interrupt system is implemented using an LSI circuit (AM-2914), which incorporates the interrupt capture flip-flops, priority encoder and interrupt reset logic into a single device.

A 4k bootstrap ROM is provided on the MC to handle the functions of automatic startup (memory verification) and bootstrap load.

5.1.4 Extended Arithmetic Unit

The EAU contains the logic necessary to support the CAU in execution of floating-point arithmetic and fixed-point multiply and divide. The module is implemented with bipolar Schottky logic.

Single precision floating-point numbers are represented in 32 bits with an 8-bit two's complement (nonbiased) exponent and a 24-bit two's complement fractional mantissa. The format is shown below:

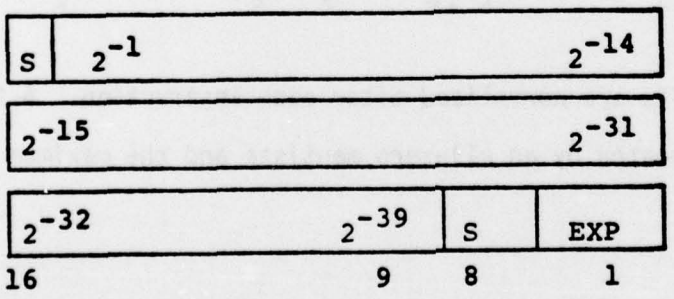


Floating-point results are normalized after each instruction. A floating-point zero is represented by an all-zero mantissa and the maximum negative exponent.

Examples of floating-point numbers:

Decimal Number	Hex Mantissa	EXP
0.5×2^{127}	4000 00	7F
0.5×2^{0}	4000 00	00
0.5×2^{-1}	4000 00	FF
0.5×2^{-128}	4000 00	80
0.0×2^{-128}	0000 00	80
-1.0×2^{127}	8000 00	3F
-1.0×2^{0}	8000 00	00
-1.0×2^{-1}	8000 00	FF
-1.0×2^{-128}	8000 00	80
-0.75×2^{-1}	A000 00	FF

Extended precision floating-point numbers are represented in 48 bits with an 8-bit two's complement exponent and a 40-bit two's complement fractional mantissa. The extended precision floating-point format is shown below:



In addition to assisting the CAU in performing floating-point calculations, the EAU provides hardware for enhancing fixed-point multiply and divide.

Figure 5-5 shows the organization of the EAU hardware. Data is exchanged with the CAU on the DI and DO buses. A 40-bit wide arithmetic unit is provided to handle floating-point mantissa and fixed-point calculations.

The X-REG is used in conjunction with the X-ROM to perform exponent calculations for floating-point operations. The X-ROM indicates a "scaling count" for mantissa alignment prior to performing a floating-point add.

The necessary control lines to coordinate the EAU data flow are derived by control ROMs (CROM) from the CAU microprogram address (CMADD) which is supplied to the EAU and by internal control ROMs.

5.1.5 I/O System

The I/O system for the Avionics Processor is composed of five separate functions:

- o BCIU-RT
- o Interrupts
- o Discretes
- o Timers
- o AGE Port

5.1.5.1 Bus Control Interface Unit (BCIU-RT)

Eight 1553 channels are provided. The BCIU-RT is the functional control unit utilized to operate and control the MIL-STD 1553A serial data interface channels. The BCIU-RT incorporates many unique features which translate into significant user advantages.

a. Microprocessor Control

- o Microprogrammed for flexibility to conform to MIL-STD-1553A, or upcoming MIL-STD-1553B (Standard Military Bus)
- o Independent of main CAU

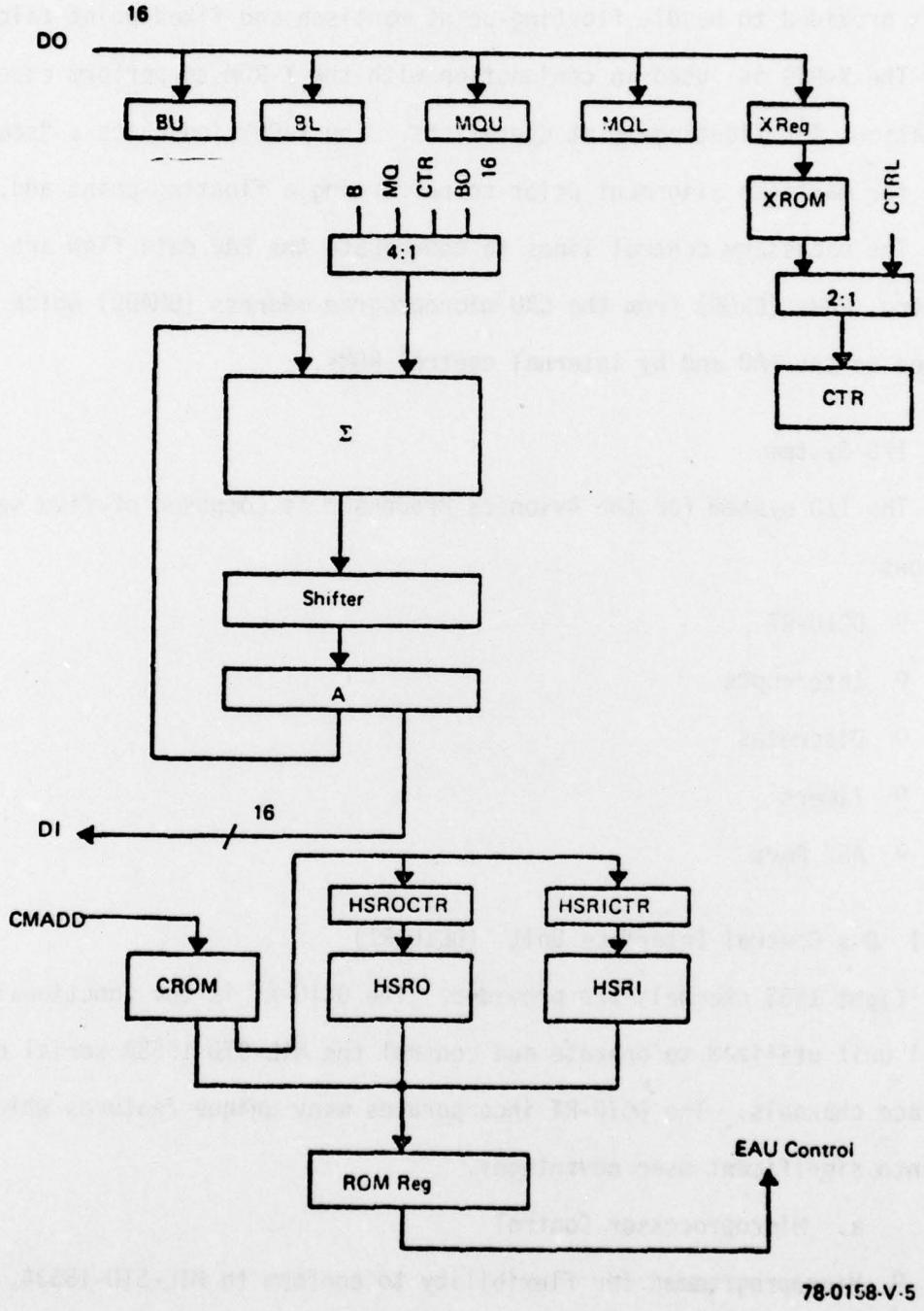


Figure 5-5. EAU Organization

- o Interfaced to main memory through DMA and RAM for minimal impact on throughput
- o No degradation of CAU throughput. CAU intervention is required only to service data channel malfunction or data channel completion or data channel interrupts.

b. Extensive Error Detection

c. Wraparound BIT

d. Self-Test

5.1.5.2 Interrupts

Sixteen levels of vectored priority interrupts are provided. Receipt of an interrupt by the CAU causes an automatic (hardware action) saving of machine status (IC, Status Word, Interrupt Mask) in memory and their subsequent replacement with a new set to accomplish interrupt vectoring. Each interrupt can be individually "masked" (with the exception of nuclear event and power down), under software control.

In schematic form, an interrupt would be handled as follows (see Figure 5-6).

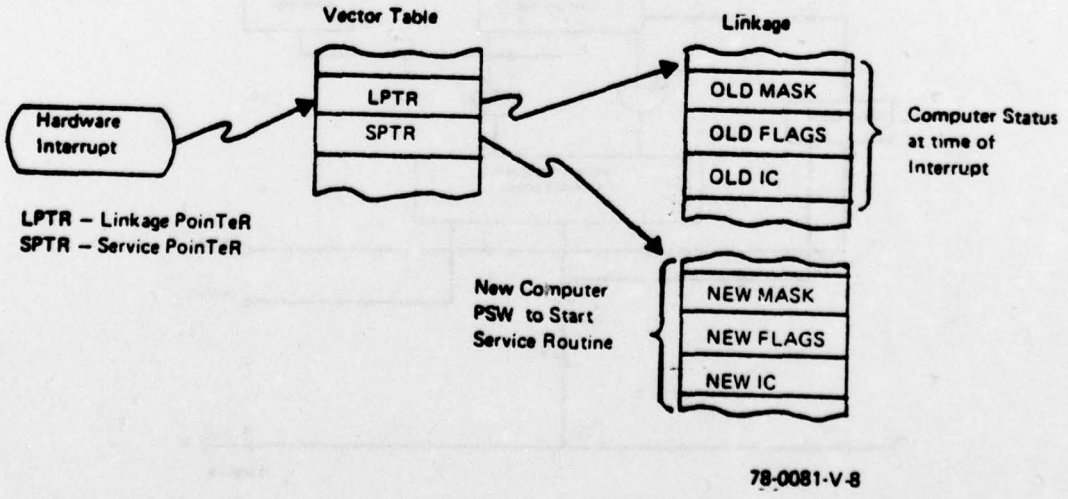


Figure 5-6. Interrupt Schematic

The "vector table" is a set of 16 double-memory words used as pointers to status word storage areas. "LPTR" (linkage pointer) points to the memory locations to receive the machine status when starting an interrupt. "SPTR" (service pointer) points to the memory locations to be used to load the new machine status which "vectors" the CAU to the interrupt service routine. Return from the interrupt service routine is accomplished by executing the LDST instruction with LPTR as an effective address. Thus the original machine status (IC, SW, Interrupt Mask) will be reinstated and the CAU will return to execute the interrupted program.

Figure 5-7 illustrates the interrupt system. System interrupts are caught in the capture register whose outputs are gated with appropriate bit in the mask register. The output of the mask gates is then encoded and presented to the CAU when INTACK is activated. The encoded interrupt number is then decoded and used to reset the appropriate capture register bit. Thus all interrupts are captured and can only be reset by servicing. Once the CAU accepts an interrupt, the interrupt system will be disabled with the exception of interrupts 14 to 16, until reactivated by a software command.

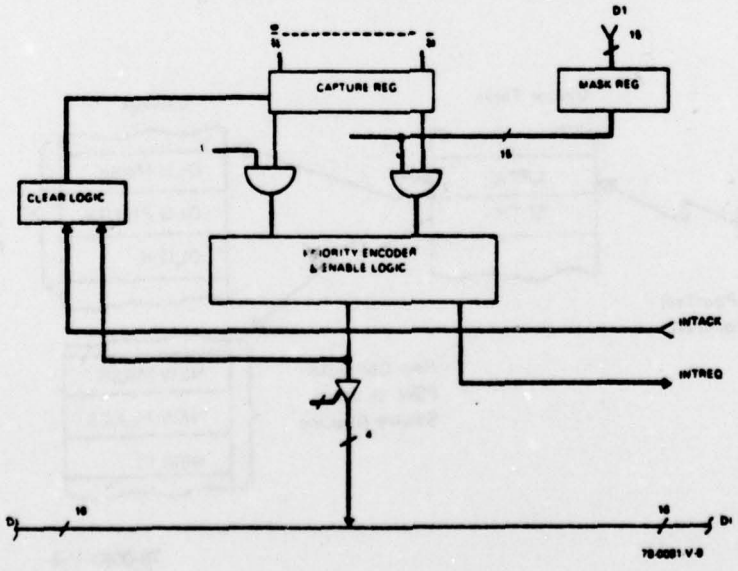


Figure 5-7. Interrupt System

The power down and nuclear event interrupts are special in that they cannot be disabled or masked off. These interrupts receive highest priority. Nuclear event interrupt differs from the other interrupts in that the hardened memory address register is stored instead of the IC to facilitate memory recovery. This interrupt also prevents the power supply from entering a false power down sequence.

Other interrupts are assigned to such events as memory protect violation, memory parity error, real-time clock or internal timer overflow, external events, etc.

The interrupt implementation scheme allows for flexibility in ordering priorities and for determining which interrupts may be masked.

Interrupt response time is a function of the instruction being executed and typically ranges from 2 to 6 usec.

5.1.5.3 Discretes

The I/O contains 32 output discretes and 32 input discretes. Output discretes are differential TTL level signals using differential line drivers. Similarly, the input discretes are received as differential TTL signals using differential line receivers. The discretes output holding register may be loaded with an OUTPUT instruction. The discrete inputs may be read by executing an INPUT instruction.

Three input discretes are dedicated to the functions CDS Load mode, Select, and Device Select. One discrete output is provided for the Processor Failure Warning. These discretes, in conjunction with an Initiate Binary Load interrupt, interface the AP with the Control and Display Subsystem (CDS).

5.1.5.4 Timers

Two programmable interval timers are provided. The timers have 100 usec per counting interval. Both interval timers count to 2^{16} and can be preset to any value in that interval. Interrupts occur when the timers overflow. The timers can also be read, started, and halted.

A real-time clock is provided that can interface with other APs. The frequency of the clock is hardware selectable. Under software control, the clock can be initialized and have its internal and external interrupts enabled or disabled.

A watchdog timer is provided which is reset by a CAU output instruction. Receipt of a reset outside the allowable window limits generates a NO-GO interrupt.

The real-time clock and one interval timer are radiation hardened.

5.1.5.5 Memory Protection

Memory write protect is accomplished by using a 1-k x 1 bit high-speed RAM under software input/output instruction control. By wiring the 10 high-order bits of the memory address and the memory read/write signal to the RAM, for example blocks of 256 memory locations can be protected by gating off the memory execute signals. This method is commonly referred to as zone protect. Larger zones can be easily accommodated by wiring only 8 bits for example. Protect zones for DMA access may also be established independently of the zones for the processor. This allows the processor access to the DMA - the processor's protected zones. Memory protect may be "overridden" from the Processor Control Unit under operator control when connected to the computer.

Figures 5-8 and 5-9 illustrate the memory zone protect concept.

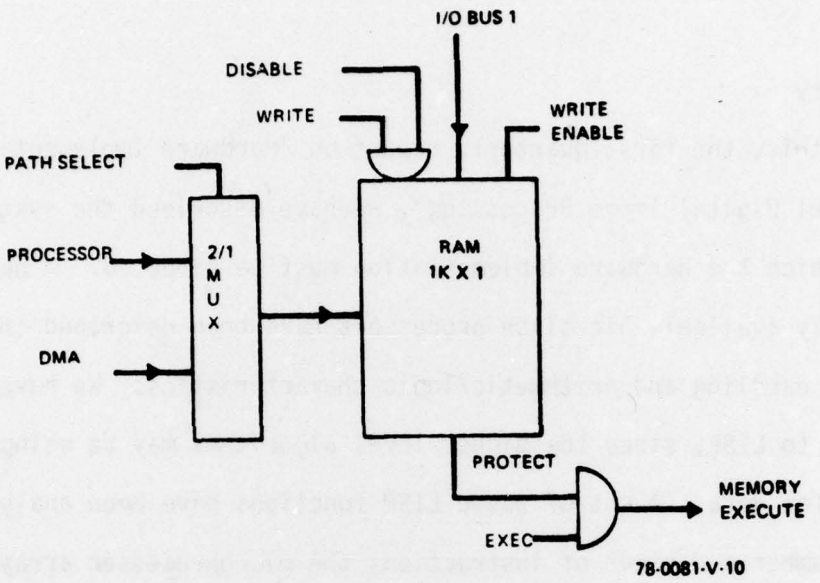


Figure 5-8. Memory Protect RAM

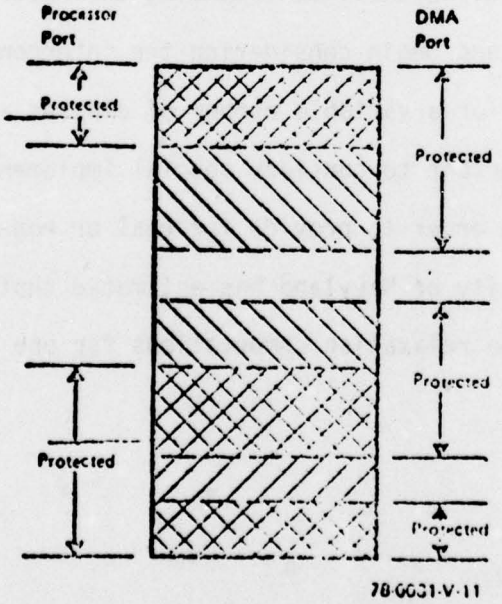


Figure 5-9. Memory Zone Protect

6.0 Summary

In this, the first quarterly report on "Hardware Implementation of - Higher Level Digital Image Processing", we have described the system design goals to which the hardware implementation must be directed. A number of commercially available bit slice processors have been described in terms of their data handling and arithmetic/logic characteristics. We have devoted some space to LISP, since the higher level algorithms may be using symbols and lists for data. A set of basic LISP functions have been analyzed to show the number and kinds of instructions the microprocessor array must accomodate to execute LISP. The relaxation algorithm which Maryland may use for higher level image processing was formulated in two versions of LISP for the discrete case. Finally an example of a Westinghouse processor was given in which bit slice microprocessors both form the ALU and control the bus.

In the next period, we shall be expanding the basic problem to 10 classes and 100 objects and begin considering the interconnect problem and dynamic reconfiguration for a variable number of classes and objects, and reliability. It is important to consider special implementation for relaxation operations in order to provide for real or non-real time operations. The University of Maryland has estimated that it will require many hours to perform the relaxation computations for one image frame on a general purpose machine.

7.0 References

1. Milgram, D.L., Rosenfeld, A., Willett, T., Tisdale, G., Algorithms and Hardware Technology for Image Recognition - Final Report, DARPA Order 3206, March 31, 1978.

2. Alexandridas, N.A., Bit - Sliced Microprocessor Architecture, Computer, Volume 11 Number 6, June 1978, IEEE Computer Society.

3. Adams, W.T., Smith, S.M., How Bit - Slice Families Compare: Part 1, Evaluating Processor Elements, Electronics, August 3, 1978, McGraw-Hill.

4. McCarthy et. al., Programming Manual for LISP 1.5, MIT Press, Cambridge, Mass. 1962.

5. Rosenfeld, A., Hummel, R.A., Zucker, S.W., Scene Labeling by Relaxation Operations, IEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-6, No. 6, June 1976.